CHAPTER 4

Introduction to Applets and Graphics

CHAPTER CONTENTS

Introduction

To this point, we've written Java applications, which run as standalone programs. Now we'll write a few Java applets, which are run by an Internet browser or an applet viewer.

As we discussed in Chapter 1, applets were originally designed to add interactivity to a web page. For example, a computer chess game on the Web can be run as an applet.

Another advantage to applets is the ease with which you can add graphics to a program. Up to this point, the input and output of our applications have been text—words and numbers. There was one exception, however: Programming Activity 1 in Chapter 3. That application opened a window and drew figures along with the text. How did we do that? We used graphics.

Graphical output is an integral part of many programs today. One compelling reason for using graphics in a program is the ability to present data in a format that is easy to comprehend. For example, our application could output average monthly temperatures as text, like this:

Jan	31
Feb	24
Mar	45
Apr	56
May	69
Jun	76
Jul	88
Aug	87
Sep	75
Oct	65
Nov	43
Dec	23

Or we could produce the bar chart shown in Figure 4.1.

The bar chart presents the same information as the text output, but it adds a visual component that makes it easier to compare the monthly temperatures—for example, to find the highest or lowest temperature or to spot temperature trends throughout the year. The colors also add information,
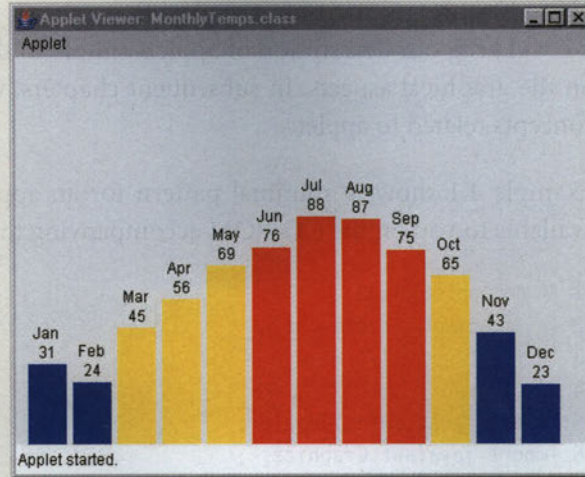
Figure 4.1

Bar Chart of Monthly Temperatures

with the low temperatures shown in blue, the moderate temperatures shown in yellow, and the high temperatures shown in red.

In this chapter, we begin by adding graphical output to applets.

4.1 Applet Structure

The *JApplet* class, an existing Java class of the *javax.swing* package, provides the basic functionality of an applet. An applet class that we write is an extension of the *JApplet* class. In Java, the *extends* keyword specifies that one class is an extension of another and *inherits* the properties of the other class. Inheritance is one of the ways to reuse classes.

An applet automatically opens a window where your program can draw shapes and text. The *main* method is not used in applets. Instead, we will use the *paint* method for our drawing code. The *paint* method is called automatically when the browser or applet viewer launches the applet, as well as any time the applet window needs to redraw itself. An applet might need to redraw itself if the user resizes the applet window or after another window, which was covering all or part of the applet window, is closed or is moved away from the applet window.

There is more to learn about applets than what is covered in this chapter. We will keep our description of applets simple so that you can concentrate on the graphical aspects. In subsequent chapters, we will cover additional concepts related to applets.

Example 4.1 shows a minimal pattern for an applet. This applet shell is available to you on the CD-ROM accompanying this book.

```
 1 /* An applet shell
 2    Anderson, Franceschi
 3 */
 4
 5 import javax.swing.JApplet;
 6 import java.awt.Graphics;
 7
 8 public class ShellApplet extends JApplet
 9 {
10    public void paint( Graphics g )
11    {
12      super.paint( g );
13      // include graphics code here
14    }
15 }
```

EXAMPLE 4.1 The *ShellApplet* Class

Lines 5 and 6 import the two classes that are used in this example: *JApplet*, used at line 8, and *Graphics*, used at line 10. The *Graphics* class is part of the *awt* (Abstract Window Toolkit) package.

Line 8 looks similar to the class header in our Java applications, but it includes two additional words: *extends JApplet*. In this case, we are inheriting from the *JApplet* class. Among other things, our *ShellApplet* class inherits the methods of the *JApplet* class. This means that we don't need to start from scratch to create an applet, so we can write applets that much faster. The *JApplet* class is called the **superclass**, and the *ShellApplet* is called the **subclass**.

The *paint* method, at lines 10–14, is where you put code to display words and graphics that should appear in the applet window. The first statement

in the *paint* method is *super.paint(g)*. This statement calls the *paint* method of our superclass, the *JApplet* class, so that it can perform its initialization of the applet window.

The *paint* method's only parameter is a *Graphics* object. This object is automatically generated by the browser or applet viewer, which sends it to the *paint* method. The *Graphics* object represents the graphics context, which, among other things, includes the applet window. The *Graphics* class contains the methods we will need to make text and shapes appear in the applet window.

Skill Practice
with these end-of-chapter questions

4.7.1 Multiple Choice Exercises

Questions 1, 2, 3, 4

4.7.4 Identifying Errors in Code

Questions 26, 27

4.7.5 Debugging Area

Question 28

4.7.8 Technical Writing

Question 38

4.2 Executing an Applet

Like applications, applets need to be compiled before they are run. Once compiled, however, applets are unlike applications in that they do not run standalone. Applets are designed to be run by an Internet browser or an applet viewer. We tell the browser to launch an applet by opening a web page that includes an *APPLET* tag as part of the HTML code. We tell the applet viewer to run the applet by specifying a minimum web page that contains an *APPLET* tag.

If you are not familiar with HTML coding, the language consists of pairs of **tags** that specify formatting for the web page. The opening tag begins the

TABLE 4.1   HTML Tags

HTML Tags	Meaning
<HTML></HTML>	Marks the beginning and end of the web page.
<HEAD></HEAD>	Marks the beginning and end of the header portion of the web page. The header contains general descriptive information about the page.
<TITLE></TITLE>	Marks the beginning and end of the text that will be displayed on the title bar of the browser or applet viewer window.
<BODY></BODY>	Marks the beginning and end of the body of the web page. The body contains the content of the web page.
<APPLET></APPLET>	Identifies the applet to launch in the browser or applet viewer window. The <APPLET> tag supports attributes for specifying the applet name, location of the class file, and size of the applet window. Each attribute consists of the attribute's name followed by an equals sign (=) and the value assigned to that attribute.
	CODE = the class name of the applet
	CODEBASE = the directory in which to search for the class file
	WIDTH = the width of the applet's window in pixels
	HEIGHT = the height of the applet's window in pixels

specific formatting; the closing tag, which is identical to the opening tag except for a leading forward slash (/), ends that formatting. The basic HTML tags used with applets are described in Table 4.1.

Example 4.2 shows a minimal HTML file that you can modify to launch an applet.

```
<HTML>
<HEAD>
   <TITLE>TitleName</TITLE>
</HEAD>
<BODY>
   <APPLET CODE="ClassName.class" CODEBASE="." WIDTH=w
           HEIGHT=h></APPLET>
</BODY>
</HTML>
```

EXAMPLE 4.2   Minimal HTML Page for Launching an Applet

The *CODE* attribute of the *APPLET* tag is the name of the applet class. The *CODEBASE* attribute is the directory in which the JVM should look for the class file. In Example 4.2, the dot (.) for the *CODEBASE* value means that the class file is in the same directory as the HTML page. The *WIDTH* and *HEIGHT* attributes specify in pixels (or picture elements) the width and height of the applet window.

For example, if we had a class called *FirstApplet*, we could use a simple text editor to create the HTML file shown in Example 4.3. In this case, the applet window will be 400 pixels wide and 300 pixels high.

```
<HTML>
<HEAD>
    <TITLE>My First Applet</TITLE>
</HEAD>
<BODY>
    <APPLET CODE="FirstApplet.class" CODEBASE="." WIDTH=400
            HEIGHT=300></APPLET>
</BODY>
</HTML>
```

EXAMPLE 4.3   HTML Page for Launching an Applet Named *FirstApplet*

An applet viewer is provided as part of Sun Microsystems' Java SE Development Kit (JDK). The applet viewer is a minimal browser that enables us to view the applet without needing to open a web browser.

If the name of the web page is *FirstApplet.html*, we can run the applet viewer from the command line as follows:

```
appletviewer FirstApplet.html
```

If you are using an Integrated Development Environment (IDE) such as TextPad, JGrasp, or Eclipse, you can run the applet viewer directly without opening a command line window. In addition, IDEs typically create a minimum web page that contains an *APPLET* tag so that you don't need to create an HTML file for each applet you write.

4.3   Drawing Shapes with *Graphics* Methods

Java's *Graphics* class, in the *java.awt* package, provides methods to draw figures such as rectangles, circles, and lines; to set the colors for drawing; and to write text in a window.

Each drawing method requires you to specify the location in the window to start drawing. Locations are expressed using an (x,y) coordinate system. Each coordinate corresponds to a pixel. The x coordinate specifies the horizontal position, beginning at 0 and increasing as you move across the window to the right. The y coordinate specifies the vertical position, starting at 0 and increasing as you move down the window. Thus for a window that is 400 pixels wide and 300 pixels high, the coordinate (0, 0) corresponds to the upper-left corner; (399, 0) is the upper-right corner; (0, 299) is the lower-left corner, and (399, 299) is the lower-right corner. Figure 4.2 shows a window with a few sample pixels and their (x,y) coordinates.

Table 4.2 shows some useful methods of the *Graphics* class for drawing shapes and displaying text in a window.

As you can see, all these methods have a *void* return type, so they do not return a value. Method calls to these methods should be standalone statements; that is, the method call should be terminated by a semicolon.

The pattern for the method names is simple. The *draw* methods render the outline of the figure, while the *fill* methods render solid figures. The *clearRect* method draws a rectangle in the background color, which effectively erases anything drawn within that rectangle.

Figure 4.3 shows the relationship among the method arguments and the figures drawn.

Figure 4.2
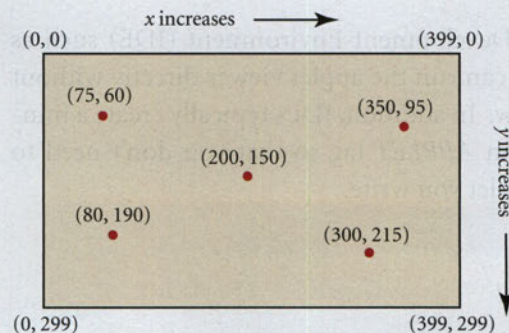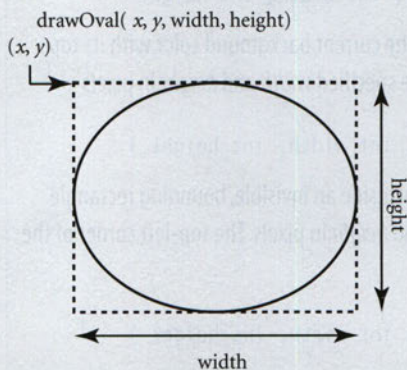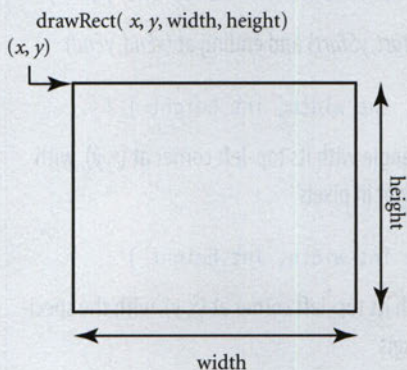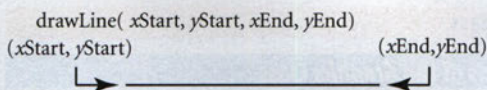
The Graphics Coordinate System

TABLE 4.2   Methods of the *Graphics* Class

Useful Methods of the *Graphics* Class	
Return value	**Method name and argument list**
void	`drawLine( int xStart, int yStart, int xEnd, int yEnd )`
	draws a line starting at (*xStart, yStart*) and ending at (*xEnd, yEnd*)
void	`drawRect( int x, int y, int width, int height )`
	draws the outline of a rectangle with its top-left corner at (*x, y*), with the specified *width* and *height* in pixels
void	`fillRect( int x, int y, int width, int height )`
	draws a solid rectangle with its top-left corner at (*x, y*), with the specified *width* and *height* in pixels
void	`clearRect( int x, int y, int width, int height )`
	draws a solid rectangle in the current background color with its top-left corner at (*x, y*), with the specified *width* and *height* in pixels
void	`drawOval( int x, int y, int width, int height )`
	draws the outline of an oval inside an invisible, bounding rectangle with the specified *width* and *height* in pixels. The top-left corner of the rectangle is (*x, y*)
void	`fillOval( int x, int y, int width, int height )`
	draws a solid oval inside an invisible, bounding rectangle with the specified *width* and *height* in pixels. The top-left corner of the rectangle is (*x, y*)
void	`drawString( String s, int x, int y )`
	displays the *String s*. If you were to draw an invisible, bounding rectangle around the first letter of the *String*, (*x, y*) would be the lower-left corner of that rectangle
void	`drawPolygon( Polygon p )`
	draws the outline of *Polygon p*
void	`fillPolygon( Polygon p )`
	draws the *Polygon p* and fills its area with the current color

drawLine(*x*Start, *y*Start, *x*End, *y*End)
(*x*Start, *y*Start)   (*x*End,*y*End)

drawRect(*x*, *y*, width, height)
(*x*, *y*)

height

width

drawOval(*x*, *y*, width, height)
(*x*, *y*)

height

width

drawString(string, *x*, *y*)

a string is written here
(*x*, *y*)

Example 4.4 shows how to use the *drawString* method. The coordinate you specify is the lower-left corner of the first character in the *String*. If you want to display more than one line of text in the default font, add 15 to the *y* value for each new line. For example, the statements at lines 13 and 14 print the message "Programming is not a spectator sport!" on two lines.
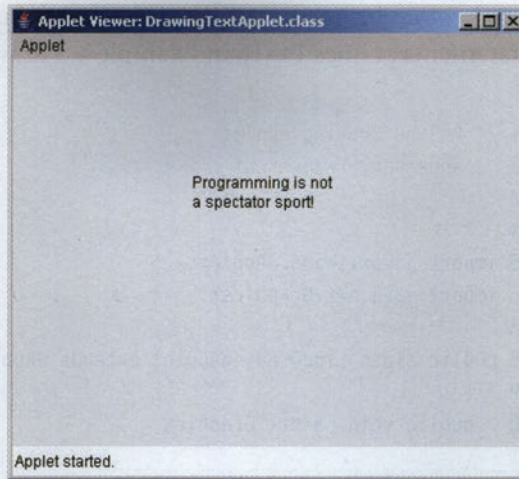
Figure 4.4

**An Applet Displaying
Two Lines of Text**

Figure 4.4 shows the output of the applet.

```
1 /* Drawing Text
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JApplet;
6 import java.awt.Graphics;
7
8 public class DrawingTextApplet extends JApplet
9 {
10   public void paint( Graphics g )
11   {
12     super.paint( g );
13     g.drawString( "Programming is not", 140, 100 );
14     g.drawString( "a spectator sport!", 140, 115 );
15   }
16 }
```

EXAMPLE 4.4   An Applet That Displays Text

To draw a line, you call the *drawLine* method with the coordinates of the beginning of the line and the end of the line. Lines can be vertical, horizontal, or at any angle. In vertical lines, the *startX* and *endX* values are the

same, while in horizontal lines, the *startY* and *endY* values are the same. Statements at lines 14–16 in Example 4.5 draw a few lines.

```
1 /* A Line Drawing Applet
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JApplet;
6 import java.awt.Graphics;
7
8 public class LineDrawingApplet extends JApplet
9 {
10   public void paint( Graphics g )
11   {
12     super.paint( g );
13
14     g.drawLine( 100, 150, 100, 250 );  // a vertical line
15     g.drawLine( 150, 75, 275, 75 );    // a horizontal line
16     g.drawLine( 0, 0, 399, 299 );      // a diagonal line from
17                                        // the upper-left corner
18                                        // to the lower-right corner
19   }
20 }
```

EXAMPLE 4.5   An Applet That Draws Lines

Figure 4.5 shows these lines drawn in an applet window.

Figure 4.5

Vertical, Horizontal, and Diagonal Lines

Example 4.6 shows how to use the methods for drawing shapes in an applet. To draw a rectangle, call the *drawRect* or *fillRect* methods with the (*x,y*) coordinate of the upper-left corner, as well as the width in pixels and the height in pixels. Obviously, to draw a square, you specify equal values for the width and height. Line 14 draws a rectangle 40 pixels wide and 100 pixels high; line 15 draws a solid square with sides that are 80 pixels in length.

Drawing an oval or a circle is a little more complex. As you can see in Figure 4.3, you need to imagine a rectangle bounding all sides of the oval or circle. Then the (*x,y*) coordinate you specify in the *drawOval* or *fillOval* method is the location of the upper-left corner of the bounding rectangle. The width and height are the width and height of the bounding rectangle. Line 17 in Example 4.6 draws a filled oval whose upper-left corner is at coordinate (100, 50) and is 40 pixels wide and 100 pixels high; this filled oval is drawn exactly inside the rectangle drawn at line 14. Line 18 draws an oval 100 pixels wide and 40 pixels high, the same dimensions as the oval drawn at line 17, but rotated 90 degrees.

You draw a circle by calling the *drawOval* or *fillOval* methods, specifying equal values for the width and height. If it seems more natural to you to identify circles by giving a center point and a radius, you can convert the center point and radius into the arguments for Java's *drawOval* or *fillOval* methods as done in lines 21–25.

```
1 /* A Shape Drawing Applet
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JApplet;
6 import java.awt.Graphics;
7
8 public class ShapeDrawingApplet extends JApplet
9 {
10   public void paint( Graphics g )
11   {
12     super.paint( g );
13
14     g.drawRect( 100, 50, 40, 100 );    // rectangle
15     g.fillRect( 200, 70, 80, 80 );     // solid square
16
17     g.fillOval( 100, 50, 40, 100 );    // oval inside the rectangle
18     g.drawOval( 100, 200, 100, 40 );   // same-size oval
19                                        // rotated 90 degrees
```
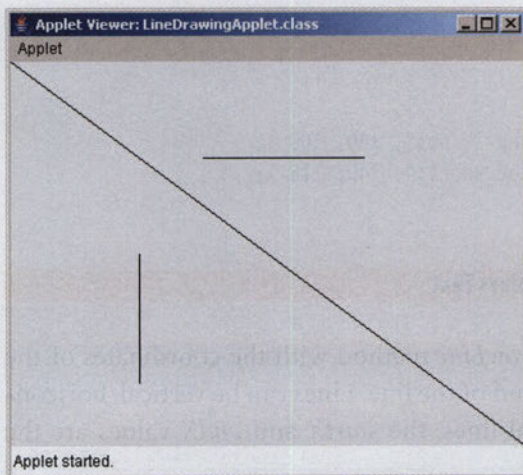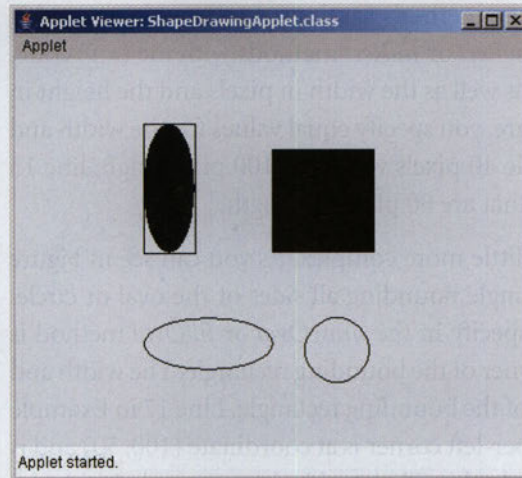
Figure 4.6

**Geometric Shapes
and Fills**


```
20
21     int centerX = 250, centerY = 225;
22     int radius = 25;
23     g.drawOval( centerX - radius, centerY - radius,
24             radius * 2, radius * 2 );  // circle using radius
25                                        // and center
26   }
27 }
```

EXAMPLE 4.6   An Applet That Draws Shapes

Figure 4.6 shows the ovals and rectangles drawn in Example 4.6.

CODE IN ACTION

To see a demonstration of the *Graphics* drawing methods, look for the Chapter 4 Flash movie on the CD-ROM accompanying this book. Click on the link for Chapter 4 to view the movie.

The *Polygon* class, which is in the *java.awt* package, allows us to draw custom shapes. The *Polygon* class represents a polygon as an ordered set of (*x,y*) coordinates; each (*x,y*) coordinate defines a vertex in the polygon. A line, called an edge, connects each (*x,y*) coordinate to the next one in the

TABLE 4.3   A Constructor and Method of the *Polygon* Class

Polygon Constructor	
`Polygon( )`	
creates an empty *Polygon*	

A Useful Method of the *Polygon* Class	
Return value	**Method name and argument list**
`void`	`addPoint( int x, int y )`
	appends the coordinate to the polygon

set. Finally, there is a line connecting the last (*x,y*) coordinate to the first one. Table 4.3 describes a constructor for the *Polygon* class, as well as a method for adding (*x,y*) coordinates to the polygon. To draw the polygon, we call the *drawPolygon* or *fillPolygon* methods of the *Graphics* class, shown in Table 4.2.

Example 4.7 demonstrates creating and drawing polygons. On line 7 we import the *Polygon* class from the *java.awt* package. On lines 15–18, we instantiate an empty *Polygon* named *triangle* and add three coordinates to it. Then we draw the triangle as an outlined polygon on line 19. On lines 21–27, we instantiate another *Polygon*, *hexagon*, and add six points to it. We draw this polygon as a solid figure on line 28. The output of this applet is shown in Figure 4.7.

```
1 /* An applet that draws polygons
2    Anderson, Franceschi
3 */
4
5 import javax.swing.JApplet;
6 import java.awt.Graphics;
7 import java.awt.Polygon;
8
9 public class DrawingPolygons extends JApplet
10 {
11   public void paint( Graphics g )
12   {
```

```
13        super.paint( g );
14
15        Polygon triangle = new Polygon( );
16        triangle.addPoint( 75, 50 );
17        triangle.addPoint( 25, 150 );
18        triangle.addPoint( 125, 150 );
19        g.drawPolygon ( triangle );
20
21        Polygon hexagon = new Polygon( );
22        hexagon.addPoint( 150, 100 );
23        hexagon.addPoint( 200, 13 );
24        hexagon.addPoint( 300, 13 );
25        hexagon.addPoint( 350, 100 );
26        hexagon.addPoint( 300, 187 );
27        hexagon.addPoint( 200, 187 );
28        g.fillPolygon ( hexagon );
29  }
30  }
```

EXAMPLE 4.7   Drawing Polygons

Figure 4.7

Output of Example 4.7
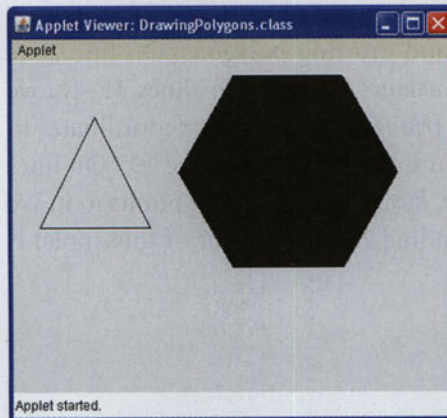
COMMON ERROR TRAP

Do not call the *paint* method. It is called automatically when the applet starts and every time the window contents need to be updated.

What happens if the (*x,y*) coordinate you specify for a figure isn't inside the window? If a figure's coordinates are outside the bounds of the window, no error will be generated, but the figure won't be visible. If the user resizes the window so that the coordinates are now within the newly sized window, then the figure will become visible.

Now we can write an applet that draws a picture. We've decided to draw an astronaut. Example 4.8 shows the code to do that. Notice that we never call

the *paint* method; it is called automatically by the applet viewer or web browser.

```
1  /* An applet with graphics
2        that draws an astronaut
3        Anderson, Franceschi
4  */
5
6  import javax.swing.JApplet;
7  import java.awt.Graphics;
8
9  public class Astronaut extends JApplet
10 {
11
12    public void paint( Graphics g )
13    {
14       super.paint( g );
15
16       int sX = 95, sY = 20; // starting x and y coordinate
17
18       // helmet
19       g.drawOval( sX + 60, sY, 75, 75 );
20       g.drawOval( sX + 70, sY + 10, 55, 55 );
21
22       // face
23       g.drawOval( sX + 83,  sY + 27, 8, 8 );
24       g.drawOval( sX + 103, sY + 27, 8, 8 );
25       g.drawLine( sX + 97, sY + 35, sX + 99, sY + 43 );
26       g.drawLine( sX + 97, sY + 43, sX + 99, sY + 43 );
27       g.drawOval( sX + 90, sY + 48, 15, 6 );
28
29       // neck
30       g.drawRect( sX + 88, sY + 70, 20, 10 );
31
32       // torso
33       g.drawRect( sX + 65, sY + 80, 65, 85 );
34
35       // arms
36       g.drawRect( sX, sY + 80, 65, 20 );
37       g.drawRect( sX + 130, sY + 80, 65, 20 );
38
39       // legs
40       g.drawRect( sX + 75, sY + 165, 20, 80 );
41       g.drawRect( sX + 105, sY + 165, 20, 80 );
```

```
42
43      // flag
44      g.drawLine( sX + 195, sY + 80, sX + 195 , sY );
45      g.drawRect( sX + 195, sY, 75, 45 );
46      g.drawRect( sX + 195, sY, 30, 25 );
47
48      // caption
49      g.drawString( "One small step for man. . .",
50                           sX + 25, sY + 270 );
51  }
52 }
```

EXAMPLE 4.8   An Applet That Draws an Astronaut

When the applet in Example 4.8 runs, our astronaut will look like the one in Figure 4.8.

To draw our astronaut, we used rectangles for the body, arms, legs, and flag; lines for the nose and the flag's stick; circles for the helmet and eyes; and an oval for the mouth. Then we used the *drawString* method to print "One small step for man..."

SOFTWARE ENGINEERING TIP

When drawing a figure using graphics, specify coordinates as offsets from a starting (*x, y*) coordinate.

In line 16, we declare and initialize two variables, *sX* and *sY*. These are the starting *x* and *y* values for the astronaut. The *x* and *y* arguments we send to the *drawRect*, *drawLine*, *drawOval*, and *drawString* methods are specified relative to this starting (*sX, sY*) coordinate. By specifying these values, such as *sX* + 60, we are using **offsets**. By using offsets from the starting (*sX, sY*) coordinate, we can easily change the position of the astronaut on the screen

Figure 4.8

An Astronaut Made from Rectangles, Ovals, Lines, and Text
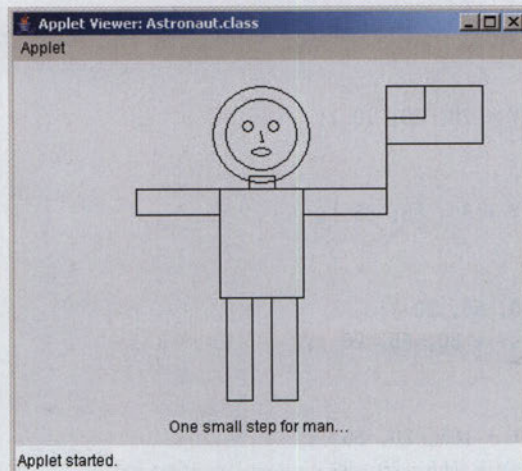
by simply changing the values of *sX* and *sY*. We don't need to change any of the arguments sent to the *Graphics* methods. To demonstrate this, try changing the values of *sX* and *sY* and re-running the applet.

> **Skill Practice**
> with these end-of-chapter questions

4.7.1 Multiple Choice Exercises

Questions 6, 7, 8, 9

4.7.2 Reading and Understanding Code

Questions 12, 13, 14, 15

4.7.3 Fill In the Code

Questions 17, 18, 19, 20

4.7.4 Identifying Errors in Code

Questions 21, 22

4.7.6 Write a Short Program

Questions 31, 32, 33

4.7.8 Technical Writing

Question 39

4.4 Using *Color*

All the figures we have drawn were black. That's because when our applet starts, the default drawing color is black. We can add color to the drawing by setting the **current color**, also called the **foreground color**, which is part of the graphics context represented by the *Graphics* object sent to the *paint* method. The *draw* and *fill* methods draw the figures in the current color. The current color remains in effect until it is set to another color. For example, if you set the current color to blue—then call the *drawRect, fillOval,* and *drawLine* methods—the rectangle, oval, and line will all be drawn in blue. Then if you set the color to yellow and call the *drawRect* method, that rectangle will be drawn in yellow.

To set the current color, use the *setColor* method of the *Graphics* class as shown in Table 4.4. This method takes a *Color* object as an argument.

TABLE 4.4   The *setColor* Method of the *Graphics* Class

Another Useful Method of the *Graphics* Class	
Return value	**Method name and argument list**
void	setColor(Color c)
	sets the current foreground color to the *Color* specified by *c*

The *Color* class, which is in the *java.awt* package, defines colors using an RGB (Red, Green, Blue) system. Any RGB color is considered to be composed of red, green, and blue components. Each component's value can range from 0 to 255; the higher the value, the higher the concentration of that component in the color. For example, a color with red = 255, green = 0, and blue = 0 is red, and a color with red = 0, green = 0, and blue = 255 is blue.

Gray consists of equal amounts of each component. The higher the value of the components, the lighter the color of gray. This makes sense because white is (255, 255, 255), so the closer a color gets to white, the lighter that color will be. Similarly, the closer the gray value gets to 0, the darker the color, because (0, 0, 0) is black.

The *Color* class provides a set of *static Color* constants representing 13 common colors. Table 4.5 lists the *Color* constants for these common colors and their corresponding red, green, and blue components.

Each color constant is a predefined *Color* object, so you can simply assign the constant to your *Color* object reference. You do not need to instantiate a new *Color* object. *Color* constants can be used wherever a *Color* object is expected. For example, this statement assigns the *Color* constant *Color.RED* to the object reference *red*:

```
Color red = Color.RED;
```

And this statement sets the current color to orange:

```
g.setColor( Color.ORANGE );
```

TABLE 4.5 *Color* Constants and Their Red, Green, and Blue Components

Color Constant	Red	Green	Blue
Color.BLACK	0	0	0
Color.BLUE	0	0	255
Color.CYAN	0	255	255
Color.DARK_GRAY	64	64	64
Color.GRAY	128	128	128
Color.GREEN	0	255	0
Color.LIGHT_GRAY	192	192	192
Color.MAGENTA	255	0	255
Color.ORANGE	255	200	0
Color.PINK	255	175	175
Color.RED	255	0	0
Color.WHITE	255	255	255
Color.YELLOW	255	255	0

In addition to using the *Color* constants, you can instantiate your own custom colors using any of the 16 million possible combinations of the component values. The *Color* class has a number of constructors, but for our purposes, we'll need only the constructor shown in Table 4.6.

TABLE 4.6 A *Color* Class Constructor

Color Constructor
Color(int rr, int gg, int bb)
allocates a *Color* object with an *rr* red component, *gg* green component, and *bb* blue component

Now let's add color to our astronaut drawing. Example 4.9 shows our modified applet.

```
1  /* An applet with graphics
2     that draws an astronaut in color
3     Anderson, Franceschi
4  */
5
6  import javax.swing.JApplet;
7  import javax.swing.JOptionPane;
8  import java.awt.Graphics;
9  import java.awt.Color;
10
11 public class AstronautWithColor extends JApplet
12 {
13
14   public void paint( Graphics g )
15   {
16     super.paint( g );
17
18     // instantiate a custom color
19     Color spacesuit = new Color( 195, 175, 150 );
20
21     int sX = 100;  // the starting x position
22     int sY = 25;   // the starting y position
23
24     // helmet
25     g.setColor( spacesuit );
26     g.fillOval( sX + 60, sY, 75, 75 );
27     g.setColor( Color.LIGHT_GRAY );
28     g.fillOval( sX + 70, sY + 10, 55, 55 );
29
30     // face
31     g.setColor( Color.DARK_GRAY );
32     g.drawOval( sX + 83,  sY + 27, 8, 8 );
33     g.drawOval( sX + 103, sY + 27, 8, 8 );
34     g.drawLine( sX + 97, sY + 35, sX + 99, sY + 43 );
35     g.drawLine( sX + 97, sY + 43, sX + 99, sY + 43 );
36     g.drawOval( sX + 90, sY + 48, 15, 6 );
37
38     // neck
39     g.setColor( spacesuit );
40     g.fillRect( sX + 88, sY + 70, 20, 10 );
41
42     // torso
```

```
43    g.fillRect( sX + 65, sY + 80, 65, 85 );
44
45    // arms
46    g.fillRect( sX, sY + 80, 65, 20 );
47    g.fillRect( sX + 130, sY + 80, 65, 20 );
48
49    // legs
50    g.fillRect( sX + 75, sY + 165, 20, 80 );
51    g.fillRect( sX + 105, sY + 165, 20, 80 );
52
53    // flag
54    g.setColor( Color.BLACK );
55    g.drawLine( sX + 195, sY + 80, sX + 195 , sY );
56    g.setColor( Color.RED );
57    g.fillRect( sX + 195, sY, 75, 45 );
58    g.setColor( Color.BLUE );
59    g.fillRect( sX + 195, sY, 30, 25 );
60
61    // caption
62    g.setColor( Color.BLACK );
63    g.drawString( "One small step for man...",
64                        sX + 25, sY + 270 );
65  }
66 }
```

EXAMPLE 4.9   An Applet That Draws an Astronaut in Color
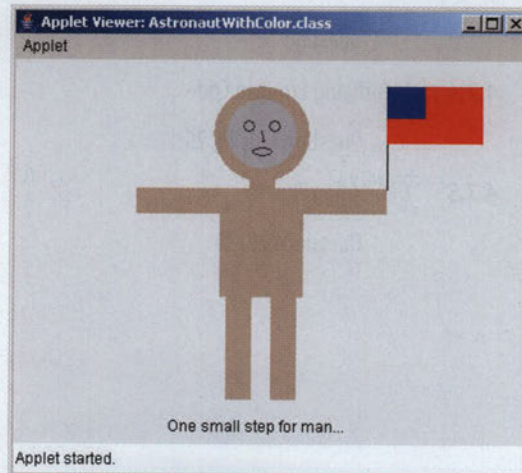
Figure 4.9 shows our astronaut in color.

Figure 4.9

Our Astronaut in Color

On line 9, we include an *import* statement for the *Color* class in the *java.awt* package.

For the space suit, we instantiate a custom *Color* object named *spacesuit* on line 19 using the constructor shown in Table 4.6. To draw the astronaut in color, we change the *draw* methods to *fill* methods, and when we draw any figure that is part of the space suit, we make sure the current color is our custom color, *spacesuit*.

It's important to realize that the rendering of the figures occurs in the order in which the *draw* or *fill* methods are executed. Any new figure that occupies the same space as a previously drawn figure will overwrite the previous figure. In this drawing, we intentionally draw the red rectangle of the flag before drawing the blue rectangle. If we drew the rectangles in the opposite order, the blue rectangle would not be visible because the red rectangle, drawn second, would cover the blue rectangle.

Skill Practice
with these end-of-chapter questions

4.7.1	Multiple Choice Exercises	
	Questions 5, 10	
4.7.2	Reading and Understanding Code	
	Question 11	
4.7.3	Fill In the Code	
	Question 16	
4.7.4	Identifying Errors in Code	
	Questions 23, 24, 25	
4.7.5	Debugging Area	
	Questions 29, 30	

4.5 Programming Activity 1: Writing an Applet with Graphics

In this Programming Activity, you will write an applet that uses graphics. You will draw a picture of your own design. The objective of this programming activity is to gain experience with the window coordinate system, the *draw* and *fill* graphics methods, and using colors.

1. Start with the *ShellApplet* class, change the name of the class to represent the figure you will draw, and add an *import* statement for the *Color* class.

2. Create a drawing of your own design. It's helpful to sketch the drawing on graph paper first, then translate the drawing into the coordinates of the applet window. Your drawing should include at least two each of rectangles, ovals, circles, and lines, plus a polygon. Your drawing should also use at least three colors, one of which is a custom color.

3. Label your drawing using the *drawString* method.

Be creative with your drawing!

? DISCUSSION QUESTIONS

1. If you define the starting (x, y) coordinate of the drawing as $(400, 400)$, you might not be able to see the drawing. Explain why and what the user can do to make the drawing visible.

2. What is the advantage to drawing a figure using a starting (x, y) coordinate?

4.6 Chapter Summary

- Applets are Java programs that are run from an applet viewer or an Internet browser. Applets are invoked via the HTML *APPLET* tag.

- When an applet begins executing, the *paint* method is called. The *paint* method is used to display text and graphics on the applet window.

- The *Graphics* class in the *java.awt* package provides methods to draw figures, such as rectangles, circles, polygons, and lines; to set the colors for drawing; and to write text in a window.

SUMMARY

- An (*x*,*y*) coordinate system is used to specify locations in the window. Each coordinate corresponds to a pixel (or picture element). The *x* value specifies the horizontal position, beginning at 0 and increasing as you move right across the window. The *y* value specifies the vertical position, starting at 0 and increasing as you move down the window.

- All drawing on a graphics window is done in the current color, which is changed using the *setColor* method.

- Objects of the *Color* class, in the *java.awt* package, can be used to set the current color. The *Color* class provides *static* constants for common colors.

- Custom *Color* objects can be instantiated by using a constructor and specifying the red, green, and blue components of the color.

4.7   Exercises, Problems, and Projects

4.7.1   Multiple Choice Exercises

1. What package does the *Graphics* class belong to?

    ❑ *Graphics*

    ❑ *java.awt*

    ❑ *swing*

    ❑ *Applet*

2. How does a programmer typically get access to a *Graphics* object when coding an applet?

    ❑ One must be created with the *Graphics* constructor.

    ❑ It is an instance variable of the class *JApplet*.

    ❑ It is a parameter of the *paint* method.

3. An applet is a standalone application.

    ❑ true

    ❑ false

4. In an applet, the *paint* method is called automatically even if the programmer does not code the method call.

    ❑ true

    ❑ false

5. Look at the following code:

```
Color c = Color.BLUE;
```

What is *BLUE*?

☐ a *static* field of the class *Color*

☐ an instance variable of the class *Color*

☐ a *static* method of the class *Color*

☐ an instance method of the class *Color*

6. What can be stated about the line drawn by the following code?

```
g.drawLine( 100, 200, 300, 200 );
```

☐ The line is vertical.

☐ The line is horizontal.

☐ The line is a diagonal.

☐ none of the above.

7. What do the arguments 10, 20 represent in the following statement?

```
g.drawRect( 10, 20, 100, 200 );
```

☐ the (x,y) coordinate of the upper-left corner of the rectangle we are drawing

☐ the width and height of the rectangle we are drawing

☐ the (x,y) coordinate of the center of the rectangle we are drawing

☐ the (x,y) coordinate of the lower-right corner of the rectangle we are drawing

8. What do the arguments 100, 200 represent in the following statement?

```
g.drawRect( 10, 20, 100, 200 );
```

☐ the (x,y) coordinate of the upper-left corner of the rectangle we are drawing

☐ the width and height of the rectangle we are drawing

☐ the height and width of the rectangle we are drawing

☐ the (x,y) coordinate of the lower-right corner of the rectangle we are drawing

9. How many arguments does the *fillOval* method take?

☐ 0

☐ 2

☐ 4

☐ 5

10. In RGB format, a gray color can be coded as *A A A* where the first *A* represents the amount of red in the color, the second *A* the amount of green, and the third *A* the amount of blue. *A* can vary from 0 to 255, including both 0 and 255; how many possible gray colors can we have?

☐ 1

☐ 2

☐ 255

☐ 256

☐ 257

4.7.2   Reading and Understanding Code

11. In what color will the rectangle be drawn?

```
g.setColor( Color.BLUE );
g.drawRect( 10, 20, 100, 200 );
```

12. What is the length of the line being drawn?

```
g.drawLine( 50, 20, 50, 350 );
```

13. What is the width of the rectangle being drawn?

```
g.fillRect( 10, 20, 250, 350 );
```

14. What is the (*x,y*) coordinate of the upper-right corner of the rectangle being drawn?

```
g.fillRect( 10, 20, 250, 350 );
```

15. What is the (*x,y*) coordinate of the lower-right corner of the rectangle being drawn?

```
g.drawRect( 10, 20, 250, 350 );
```

4.7.3 Fill In the Code

16. This code sets the current color to red.

```
// assume you have a Graphics object named g
// your code goes here
```

17. This code draws the *String* "Fill in the Code" with the lower-left corner of the first character (the *F*) being at the coordinate (100, 250).

```
// assume you have a Graphics object called g
// your code goes here
```

18. This code draws a filled rectangle with a width of 100 pixels and a height of 300 pixels, starting at the coordinate (50, 30).

```
// assume you have a Graphics object called g
// your code goes here
```

19. This code draws a filled rectangle starting at (50, 30) for its upper-left corner with a lower-right corner at (100, 300).

```
// assume you have a Graphics object called g
// your code goes here
```

20. This code draws a circle of radius 100 with its center located at (200, 200).

```
// assume you have a Graphics object called g
// your code goes here
```

4.7.4 Identifying Errors in Code

21. Where is the error in this code sequence?

```
Graphics g = new Graphics( );
```

22. Where is the error in this code sequence?

```
// we are inside method paint
g.drawString( 'Find the bug', 100, 200 );
```

23. Where is the error in this code sequence?

```
// we are inside method paint
g.setColor( GREEN );
```

24. Where is the error in this code sequence?

```
// we are inside method paint
g.setColor( Color.COBALT );
```

25. Where is the error in this code sequence?

```
// we are inside method paint
g.color = Color.RED;
```

26. Where is the error in this statement?

```
import Graphics;
```

27. Where is the error in this statement?

```
import java.awt.JApplet;
```

4.7.5 Debugging Area—Using Messages from the Java Compiler and Java JVM

28. You coded the following program in the file *MyApplet.java*:

```java
import javax.swing.JApplet;
import java.awt.Graphics;

public class MyApplet extends JApplet
{
    public static void paint( Graphics g )     // line 6
    {
        // some code here
    }
}
```

When you compile, you get the following message:

```
MyApplet.java:6: paint(Graphics) in MyApplet cannot
override paint(Graphics) in Container;
 public static void paint( Graphics g )     // line 6
                 ^

 overriding method is static
1 error
```

Explain what the problem is and how to fix it.

29. You imported the *Color* class and coded the following on line 10 of the class *MyApplet.java*:

```java
Color c = new Color( 1.4, 234, 23 );     // line 10
```

When you compile, you get the following message:

```
MyApplet.java:10: cannot find symbol

Color c = new Color( 1.4, 234, 23 );     // line 10
                 ^

symbol  : constructor Color (double,int,int)
location : class Color
1 error
```

Explain what the problem is and how to fix it.

30. You coded the following on line 10 of the class *MyApplet.java*:

```
Color c = Color.Blue; // line 10
```

When you compile, you get the following message:

```
MyApplet.java:10: cannot find symbol
     Color c = Color.Blue;    // line 10
                     ^
```

```
symbol  : variable Blue
location: class Color
1 error
```

Explain what the problem is and how to fix it.

4.7.6 Write a Short Program

31. Write an applet that displays the five Olympic rings.

32. Write an applet that displays a tic-tac-toe board. Include a few X's and O's.

33. Write an applet that displays a rhombus (i.e., a parallelogram with equal sides). Your rhombus should not be a square.

4.7.7 Programming Projects

34. Write an applet that displays two eyes. An eye can be drawn using an oval, a filled circle, and lines. On the applet, write a word or two about these eyes.

35. Write an applet that displays the following coins: a quarter, a dime, and a nickel. These three coins should be drawn as basic circles (of different diameters) with the currency value inside (for instance, "$.25").

36. Write an applet that displays a basic house, made up of lines (and possibly rectangles). Your house should have multiple colors. On the applet, give a title to the house (for instance, "Java House").

37. Write an applet that displays a black and red bull's eye target, typically made up of several concentric circles.

4.7.8   Technical Writing

38. On the World Wide Web, an applet is a program that executes on the "client side" (a local machine such as your own PC) as opposed to the "server side" (such as a server at *www.yahoo.com*). Do you see any potential problem executing the same program, such as an applet, on possibly millions of different computers worldwide?

39. If the *drawRect* method did not exist, but you still had the *drawLine* method available, explain how you would be able to draw a rectangle.

4.7.9   Group Project (for a group of 1, 2, or 3 students)

40. Write an applet and one HTML file calling the applet.

    The applet should include the following:

    ❑ a drawing of a chessboard piece (it can be in a single color)

    ❑ a description of a particular piece of a chessboard (for instance, a rook) and its main legal moves

    In order to make the description visually appealing, you should use several colors and several fonts. You will need to look up the following on Sun's Java website:

    ❑ the *Font* class

    ❑ how the *Font* class constructors work

    ❑ the method *setFont* of the *Graphics* class